Preprint
UCRL-JC-134897

Climate Data Management System

R. Drach

This article was submitted to
American Meteorological Society
11th Symposium on Global Change Studies
Long Beach, CA
January 9-14, 2000

July 13, 1999

U.S. Department of Energy
Lawrence Livermore National Laboratory

Climate Data Management System

Robert Drach

Program for Climate Model Diagnosis and Intercomparison

Lawrence Livermore National Laboratory

March 1999

CHAPTER 1

Introduction

1.1 Overview

The Climate Data Management System is an object-oriented data management system, specialized for organizing multidimensional, gridded data used in climate analysis and simulation.

1.2 Basic Concepts

The building blocks of CDMS are variables, container classes, structural classes, and links. All gridded data stored in CDMS is associated with variables. The container objects group variables and structural objects. Variables are defined in terms of structural objects.

Most CDMS objects can have *attributes*, which are scalar or one-dimensional metadata items. Attributes which are stored in the database, that is are persistent, are called *external* attributes. Some attributes are *internal*: they are associated with an object but do not appear explicitly in the database.

1.2.1 Variables

Most of the data stored in CDMS has the form of multidimensional data arrays. A *variable* is a persistent, multidimensional array, together with associated metadata. A persistent variable retains its value independent of an application.

A variable may be viewed as a function which maps a multidimensional *domain* onto a range of values. The domain of a variable consists of an ordered tuple of axes and/or grids which define the shape and spatial orientation of the variable.

1.2.2 Container classes: Databases, Datasets, and CdmsFiles

Variables are contained in *datasets*. A *dataset* is a collection of variables and associated structural objects. All objects in a dataset are identified by a string ID, unique within the dataset.

The data contained in a dataset generally is stored in one or more physical datafiles. An additional ASCII metafile describes how the files are organized and named. In a climate simulation application, a dataset usually represents the data generated by one run of a general circulation or coupled ocean-atmosphere model.

The metafile associated with a dataset can contain information which is additional to that in the actual data files. The format of the metafile is designed for readability, ease of extension, and integration with Web browsers. It can be used to enforce naming standards, by 'aliasing' variable names. The format of the metafile is based on the World Wide Web Council standard XML language (see Chapter 5).

A *Database* is a collection of datasets and other CDMS objects. A Database:

- provides naming mechanisms for accessing and searching its contents independent of local file names.

- may be associated with a server, local or remote. A given site would ordinarily have only a small number of Databases, perhaps one public and a few private ones.
- provides a facility for standardization of data. The objects contained in the Database can be required to adhere to a metadata standard such as GDT. This provides assurance that data access will be robust.

The process of copying external data into a Database is known as the *ingest* process. Ingesting data into CDMS involves verifying that data adheres to a standard. Mechanisms are provided for adding metadata to meet that standard. The *cdingest* utility is used to ingest data into CDMS (See Section 6.1).

CDMS permits access to data files which are 'outside' a database. In CDMS, a file is termed a *CdmsFile*. CdmsFiles are similar to datasets, in that they are containers for variables, axes, and grids. However, not all CDMS objects can be stored in CdmsFiles. Also, the standardization associated with the ingest process may not apply to a CdmsFile. Data may be read from a variety of self-describing file formats, including netCDF, HDF, GRIB, and PCMDI DRS formats.

1.2.3 Structural classes: Axes and Grids

Structural objects are used in the definition of variables. They define how a variable is oriented in space and time. For example, suppose that a variable is a function of time, longitude, and latitude. The domain of the variable consists of an ordered tuple of *axes* (time, longitude, and latitude). The domain ordering corresponds to the physical ordering of data: the first axis is 'slowest varying'.

An axis is a one-dimensional coordinate vector. Within a dataset, an axis may be shared by more than one variable. An axis may identified as spatio-temporal: a time, vertical level, latitude, or longitude axis.

CDMS allows generalization of domains to include *grids*. In spatial terms, a *grid* is a horizontal partitioning of all or a portion of the Earth's surface. A grid which can be represented as a pair of axes (latitude, longitude) is called a *RectGrid*. For example, if the domain of a variable is (time, latitude, longi-

1.3.2 Partition

One more piece of information is required to fully describe the dataset partitioning: the partition attribute. Each axis which is partitioned has a **.partition** attribute, which is a list of the start and end indices of each axis partition.

FIGURE 1. Partitioned axis

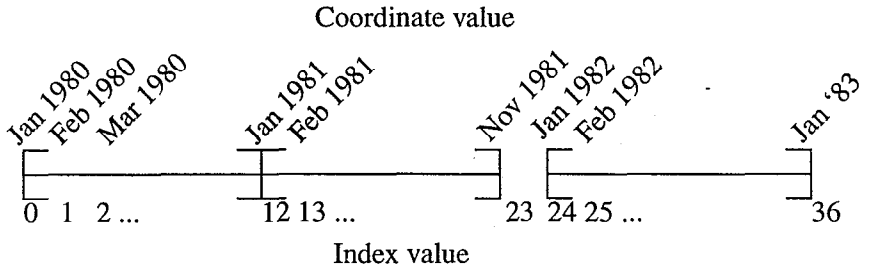

Figure 1 shows a time axis, representing the 36 months January, 1980 through December, 1982, with December 1981 missing. The first partition interval is (0,12), the second is (12,23), and the third is (24,36), where the interval (i,j) represents all indices k such that i <= k < j. The .partition attribute for this axis would be the list:

```
[0, 12, 12, 23, 24, 36]
```

Note that the end index of the second interval is strictly less than the start index of the following interval. This indicates that data for that period is missing.

CHAPTER 2

CDMS Python Application Programming Interface

2.1 Overview

This chapter describes the CDMS Python application programming interface (API). Python is a popular public-domain, object-oriented language. Its features include support for object-oriented development, a rich set of programming constructs, and an extensible architecture. CDMS itself is implemented in a mixture of C and Python. In this chapter the assumption is made that the reader is familiar with the basic features of the Python language.

Python supports the notion of a **module**, the biggest program unit in the language. Modules group together associated classes and methods, and provide a separate namespace. The **import** command makes the module accessible to an application. This chapter documents the **cdms** module.

The chapter sections correspond to the CDMS classes. Each section contains tables describing the class internal (non-persistent) attributes, constructors (functions for creating an object), and class methods. Method datatypes may be any of the Python types:

Table 2.1 Python types used in CDMS

Type	Description
Array	Numeric multidimensional data array. All elements of the array are of the same type. Defined in the **Numeric** module.
Comptime	Absolute time value, a time with representation (year, month, day, hour, minute, second). Defined in the **cdtime** module. cf. **reltime**
Dictionary	An unordered collection of objects, indexed by key. All dictionaries in CDMS are indexed by strings, e.g.: `axes['time']`
Float	Floating-point value.
Integer	Integer value.
List	An ordered sequence of objects, which need not be of the same type. New members can be inserted or appended. Lists are denoted with square brackets, e.g., `[1, 2.0, 'x', y']`
None	No value returned.
Reltime	Relative time value, a time with representation (value, "units since basetime"). Defined in the **cdtime** module. cf. **comptime**
Tuple	An ordered sequence of objects, which need not be of the same type. Unlike lists, tuples elements cannot be inserted or appended. Tuples are denoted with parentheses, e.g., `(1, 2.0, 'x', y')`

2.2 *A first example*

The following Python script reads January and July monthly temperature data from an input dataset, averages over time, and writes the results to an output file. The input temperature data is ordered (time, latitude, longitude).

```
 1  #!/usr/local/bin/python
 2  import cdms, Numeric
 3  jones = cdms.openDataset('/pcmdi/cdms/obs/jones_mo.xml','r')
 4  tasvar = jones.variables['tas']
 5  jans = tasvar[0::12]
 6  julys = tasvar[6::12]
 7  janavg = Numeric.avg.reduce(jans)
 8  julyavg = Numeric.avg.reduce(julys)
 9  out = cdms.createDataset('janjuly.nc')
10  grid = tasvar.getGrid()
11  outgrid = out.copyGrid(grid)
12  janvar = out.createVariable('tas_jan', cdms.CdFloat,
        (outgrid,))
13  julyvar = out.createVariable('tas_jul', cdms.CdFloat,
        (outgrid,))
14  janvar.units = julyvar.units = "K"
15  janvar.long_name = "mean January surface temperature"
16  julyvar.long_name = "mean July surface temperature"
17  janvar[:] = janavg
18  julyvar[:] = julyavg
19  jones.close()
20  out.close()
```

Line	Notes
2	Makes the CDMS and Numeric modules available.
3	Opens the input dataset, read-only. The `.xml` file is an ASCII file which describes the data files in the dataset. The result `jones` is a dataset object.

Line	Notes
4	Gets the surface air temperature variable. `'tas'` is the name of the variable in the input dataset. `jones.variables` is a Python dictionary, which maps the variable name (`'tas'`) to the variable object (`tasvar`).
5	Reads all January monthly mean data into a Numeric array `jans`. Variables can be sliced as if they were Numeric arrays. The slice operator `[0::12]` means 'take every 12th slice from dimension 0, starting at index 0 and ending at the last index.' If the stride `12` were omitted, it would default to 1. Note that the variable is actually 3-dimensional. Since no slice is specified for the second or third dimensions, all values of those dimensions are retrieved. The slice could also have been written `[0::12, : , :]`. Also note that the data may be read from multiple data files, depending on the organization of the dataset. CDMS opens the needed data files, extracts the appropriate slices, and concatenates them into the result array as necessary.
6	Reads all July data into a Numeric array `julys`.
7	Averages `jans` across the first array dimension, time. The result is a function of latitude and longitude.
8	Averages `julys` across time.
9	Creates a new netCDF output file named `'janjuly.nc'` to hold the results.
10	Gets the grid object `grid` associated with `tasvar`, contained in the input dataset.
11	Copies `grid` to the output file. `outgrid` is a grid object contained in the output file.

Line	Notes
12	Creates a variable `janvar` in the output file, as a function of the output grid. Its identifier in the output file is the string `'tas_jan'`. The last argument is a tuple of the grids and/or axes which define the domain of the variable. Note that as yet no data has been written to the file.
13	Creates a new variable `julyvar` in the output file.
14	Creates a `.units` attribute for both variables, and writes the string value `'K'` to the output file. There is nothing special about `.units`; any attribute can be created and written in similar fashion. Global attributes are written by setting an attribute of the file object.
17	Writes the January average data to the output file. Setting a slice of a variable writes data to that variable. In this case, the slice [:] references all data for the variable.
18	Writes July average data to the output file.
19	Closes the input dataset.
20	Closes the output file.

2.3 *cdms module*

The **cdms** module is the Python interface to CDMS. The objects and methods in this chapter are made accessible with the command:

```
import cdms
```

The functions described in this section are not associated with a class. Rather, they are called as module functions, e.g.,

```
file = cdms.createDataset('sample.nc')
```

Table 2.2 cdms module functions

Type	Definition
Axis	**createAxis(data, bounds=None):** Create an Axis, which is not associated with a file or dataset. This is useful for creating a grid which is not contained in a file or dataset. *data* is a one-dimensional, monotonic Numeric array. *bounds* is an array of shape (len(data),2), such that for all i, data[i] is in the range [bounds[i,0],bounds[i,1]]. Also see: CdmsFile.createAxis
Dataset or CdmsFile	**createDataset(path,template=None)** Create a Dataset or CdmsFile. **(Note: Only CdmsFile creation is implemented at present.)** *path* is the XML file name, or netCDF filename for simple file create. If the path extension is '.xml' or '.cdml', a Dataset is created. Otherwise a netCDF CdmsFile is created. *template* is a string filename template for the datafile(s), for dataset creation. This argument should be omitted for CdmsFile creation. **Example**: Create a new netCDF file. `file = cdms.createDataset('sample.nc')`
Axis	**createEqualAreaAxis(nlat)** Create an equal-area latitude axis. The latitude values range from north to south, and for all axis values x[i], sin(x[i])-sin(x[i+1]) is constant. *nlat* is the axis length. The axis is not associated with a file or dataset.

Table 2.2 cdms module functions

Type	Definition
Axis	**createGaussianAxis(nlat)** Create a Gaussian latitude axis. Axis values range from north to south. *nlat* is the axis length. The axis is not associated with a file or dataset.
RectGrid	**createGenericGrid(latArray, lonArray, latBounds=None, lonBounds=None, order="yx", mask=None)** Create a generic grid, that is, a grid which is not typed as Gaussian, uniform, or equal-area. The grid is not associated with a file or dataset. *latArray* is a NumPy array of latitude values. *lonArray* is a NumPy array of longitude values *latBounds* is a NumPy array having shape (len(latArray),2), of latitude boundaries. *lonBounds* is a NumPy array having shape (len(lonArray),2), of longitude boundaries. *order* is a string specifying the order of the axes, either "yx" for (latitude, longitude), or "xy" for the reverse. *mask* (optional) is an integer-valued NumPy mask array, having the same shape and ordering as the grid.
RectGrid	**createGlobalMeanGrid(grid)** Generate a grid for calculating the global mean via a regridding operation. The return grid is a single zone covering the range of the input grid. *grid* is a RectGrid.

Table 2.2 cdms module functions

Type	Definition
RectGrid	**createRectGrid(lat, lon, order, type="generic", mask=None)** Create a rectilinear grid, not associated with a file or dataset. This might be used as the target grid for a regridding operation. *lat* is a latitude axis, created by cdms.createAxis. *lon* is a longitude axis, created by cdms.createAxis. *order* is a string with value "yx" (the first grid dimension is latitude) or "xy" (the first grid dimension is longitude). *type* is one of 'gaussian','uniform','equalarea',or 'generic' If specified, *mask* is a two-dimensional, logical Numeric array (all values are zero or one) with the same shape as the grid.
RectGrid	**createUniformGrid(startLat, nlat, deltaLat, startLon, nlon, deltaLon, order="yx", mask=None)** Create a uniform rectilinear grid. The grid is not associated with a file or dataset. The grid boundaries are at the midpoints of the axis values. *startLat* is the starting latitude value. *nlat* is the number of latitudes. *deltaLat* is the increment between latitudes. *startLon* is the starting longitude value. *nlon* is the number of longitudes. *deltaLon* is the increment between longitudes. *order* is a string with value "yx" (the first grid dimension is latitude) or "xy" (the first grid dimension is longitude). If specified, *mask* is a two-dimensional, logical Numeric array (all values are zero or one) with the same shape as the grid.

Table 2.2 cdms module functions

Type	Definition
Axis	**createUniformLatitudeAxis(startLat, nlat, deltaLat)** Create a uniform latitude axis. The axis boundaries are at the midpoints of the axis values. The axis is designated as a circular latitude axis. *startLat* is the starting latitude value. *nlat* is the number of latitudes. *deltaLat* is the increment between latitudes.
RectGrid	**createZonalGrid(grid)** Create a zonal grid. The output grid has the same latitude as the input grid, and a single longitude. This may be used to calculate zonal averages via a regridding operation. *grid* is a RectGrid.
Axis	**createUniformLongitudeAxis(startLon, nlon, deltaLon)** Create a uniform longitude axis. The axis boundaries are at the midpoints of the axis values. The axis is designated as a circular longitude axis. *startLon* is the starting longitude value. *nlon* is the number of longitudes. *deltaLon* is the increment between longitudes.

Table 2.2 cdms module functions

Type	Definition
List	**matchPattern(objlist, pattern, attname=None, classtag=None)** Search an object or list of objects for a string attribute with value that matches a pattern. The return value is a list of objects having a matching attribute. If no objects are found, an empty list is returned. The pattern must match the entire attribute value. To match a substring, use **searchPattern**. *objlist* is a CdmsObj or a list of CdmsObj (typically a list of Datasets and/or CdmsFiles). Searching a Dataset searches all objects contained in the Dataset as well. *pattern* is a regular expression, as defined in the Python **re** module. The pattern must match the entire attribute value. To search for a substring, see **searchPattern**. If *attname* is specified, the search is restricted to string-valued attributes with that name. If *classtag* is specified, the search is restricted to objects with the given classtag. See Table 2.3 on page 27 for a list of classtags. **Example**: Find all objects with boundaries. `matches = cdms.matchPattern(datasets,"bounds.*")`

Table 2.2 **cdms module functions**

Type	Definition
Dataset or CdmsFile	**openDataset(url,mode='r',template=None)** Open or create a Dataset or CdmsFile. *url* is a Uniform Resource Locator, referring to a cdunif or XML file. If the URL has the extension '.xml' or '.cdml', a Dataset is returned, otherwise a CdmsFile is returned. If the URL protocol is 'http', the file must be a '.xml' or '.cdml' file, and the mode must be 'r'. If the protocal is 'file' or is omitted, a local file or dataset is opened. *mode* is the open mode. See Table 2.16 on page 43. *template* is a string file template for the datafile(s), for dataset creation. This argument should be omitted except for Dataset creation. **Example:** `f = cdms.openDataset("sampleset.xml")`

Table 2.2 cdms module functions

Type	Definition
Integer	**searchPattern(objlist, pattern, attribute=None, tag=None)** Search an object or list of objects for a string attribute which has a substring matching *pattern*. To match the entire attribute value, use **matchPattern**. *objlist* is a CdmsObj or a list of CdmsObj objects (typically a list of Datasets and/or CdmsFiles). Searching a Dataset searches all objects contained in the Dataset as well. *pattern* is a regular expression, as defined in the Python **re** module. The pattern must match the entire attribute value. To search for a substring, see **searchPattern**. If *attname* is specified, the search is restricted to string-valued attributes with that name. If *classtag* is specified, the search is restricted to objects with the given classtag. See Table 2.3 on page 27 for a list of classtags. **Example**: Find all COARDS datasets: `matches = cdms.searchPattern(datasets,"COARDS","Conventions","dataset")`

Table 2.2 **cdms module functions**

Type	Definition

Integer	**searchPredicate(objlist, predicate, classtag=None)**
	
	Search an object or list of objects for those objects which satisfy a predicate condition. The function returns a list of objects for which the predicate is true, or the empty list if no such object is found.
	
	objlist is a CdmsObj (see "CdmsObj" on page 29) or a list of CdmsObj (typically a list of Datasets and/or CdmsFiles). Searching a Dataset searches all objects contained in the Dataset as well.
	
	predicate is a function which takes a single object (dataset, variable, axis, etc.) and returns true or false. Lambda functions are useful for defining predicates. For example, to search for objects with partition length greater than 1000, set the predicate to
	
	`lambda obj: obj.partition_length > 1000`
	
	Note that Attribute errors are ignored. This simplifies building predicate functions, as it is not necessary to test whether the object has a particular attribute. In the previous example, it is unnecessary to test for the existence of the partition_length attribute.
	
	If *classtag* is specified, only those objects of the given class are searched. Class tags are defined in Table 2.3 on page 27.
	
	Example: Find all datasets with a variable named "hfss":
	
	`f = lambda obj:obj.variables.has_key("hfss")`
	`matches =`
	`cdms.searchPredicate(somelist,f,"dataset")`

Table 2.2 cdms module functions

Type	Definition
None	**setAutoBounds(mode)** Set autobounds mode. If *mode* is 'on' (the default), getBounds will automatically generate boundary information for a grid, if the boundaries are not explicitly defined. If *mode* is 'off', and no boundary data is explicitly defined, the bounds will NOT be generated; getBounds will return None for the boundaries.
None	**setAutoReshapeMode(mode)** If *mode* is 'on', remove singleton dimensions from all arrays returned by slice operators or **Variable.getRegion()**. The default mode is 'off'. The autoreshape mode may be changed at any time. Note: Enabling autoreshape causes non-contiguous arrays to be copied to contiguous arrays. This will incur some CPU overhead. **Example**: Enable autoreshape mode. `cdms.setAutoReshapeMode('on')`
None	**setClassifyGrids(mode)** Set grid classifiction mode. This affects how grid type is determined, for the purpose of generating grid boundaries. If mode is 'on' (the default), grid type is determined by a grid classification method, regardless of the value of grid.getType(). If mode is 'off', the value of grid.getType() determines the grid type

Table 2.3 **Class Tags**

Tag	Class
'axis'	Axis
'dataset'	Dataset, CdmsFile
'grid'	RectGrid
'variable'	Variable
'xlink'	Xlink

2.3.1 Example: Searching a list of datasets

Given the following definitions:

```
# Print a list of objects returned from a search
def printmatches(title,matches):
    print '\n',title
    for obj in matches:
        if hasattr(obj,'uri'):
            path = obj.uri
        else:
            path = obj.parent.id
        print path, obj.id

# Return the shape of an object's grid
def gridshape(obj):
    latlen = lonlen = 0
    for axis,start,length,true_length in obj.domain:
        if axis.isLatitude(): latlen = length
        if axis.isLongitude(): lonlen = length
    return (latlen,lonlen)

# The list of datasets to search
paths = [
    '/pcmdi/drach/xml/sample/obs/ncep_reanalysis_6h.xml',
    '/pcmdi/drach/xml/sample/obs/ncep_reanalysis_6h_multi.xml',
    '/pcmdi/drach/xml/sample/cmip2/ccc/perturb.xml',
    '/pcmdi/drach/cdms/src/python/test/testgdtrel.nc'
    ]

# Generate a list of dataset objects
datasets = map(cdms.openDataset,paths)
```

Find objects with bounds defined:

```
matches = cdms.searchPattern(datasets,".*","bounds")
printmatches('Objects with boundaries',matches)
```

List all COARDS datasets:

```
matches =
    cdms.searchPattern(datasets,"COARDS","Conventions","dataset")
printmatches('Coards datasets',matches)
```

Find all axes with gaps:

```
f = lambda obj:obj.length!=obj.partition_length
matches = cdms.searchPredicate(datasets,f,"axis")
printmatches('Axes with gaps',matches)
```

Find all datasets with a variable named "hfss":

```
f = lambda obj:obj.variables.has_key("hfss")
matches = cdms.searchPredicate(datasets,f,"dataset")
printmatches('Datasets with variable "hfss"',matches)
```

Find all variables on 32x64 grids:

```
f = lambda obj: gridshape(obj)==(32,64)
matches = cdms.searchPredicate(datasets, f, "variable")
printmatches('Variables on 32x64 grids',matches)
```

Find all axes with length greater than 1000:

```
f = lambda obj: obj.length>1000
matches = cdms.searchPredicate(datasets, f, "axis")
printmatches('Axes with length>1000',matches)
```

Find all axes with length one:

```
f = lambda obj: len(obj)==1
matches = cdms.searchPredicate(datasets, f, "axis")
printmatches('Axes with length 1',matches)
```

Find boundary arrays:

```
matches = cdms.matchPattern(datasets, "bounds.*")
printmatches('Objects with boundaries',matches)
```

2.4 CdmsObj

A CdmsObj is the base class for all CDMS database objects. At the application level, CdmsObj objects are never created and used directly. Rather the subclasses of CdmsObj (Dataset, Variable, Axis, etc.) are the basis of user application programming.

All objects derived from CdmsObj have a special attribute **.attributes**. This is a Python dictionary, which contains all the external (persistent) attributes associated with the object. This is in contrast to the internal, non-persistent attributes of an object, which are built-in and predefined.

Example: get a list of all external attributes of obj.

```
extatts = obj.attributes.keys()
```

Table 2.4 Attributes common to all CDMS objects

Type	Name	Definition
Dictionary	attributes	External attribute dictionary for this object.

All attributes may be accessed and set using the Python dot notation ('.')

Table 2.5 Getting and setting attributes

Type	Definition
Various	**value = obj.attname** Get an internal or external attribute value. If the attribute is external, it is read from the database. If the attribute is not already in the database, it is created as an external attribute. Internal attributes cannot be created, only referenced. **obj.attname = value** Set an internal or external attribute value. If the attribute is external, it is written to the database.

2.5 *Axis*

An Axis is a one-dimensional coordinate object.

An Axis is contained in a Dataset. Setting a slice of an Axis writes data to the Dataset, referencing an Axis slice reads data from the Dataset. Axis objects are also used to define the domain of a Variable.

An axis in a CdmsFile may be designated the 'unlimited' axis, meaning that it can be extended in length after the initial definition. There can be at most one unlimited axis associated with a CdmsFile.

Table 2.6 Axis Internal Attributes

Type	Name	Definition
Dictionary	attributes	External attribute dictionary.

Table 2.6 Axis Internal Attributes

Type	Name	Definition
String	id	Axis identifer.
Dataset	parent	The dataset which contains the variable.
Tuple	shape	The length of each axis.

Table 2.7 Axis Constructors

cdms.createAxis(data, bounds=None)

Create an axis which is not associated with a dataset or file. See Table 2.2 on page 18.

Dataset.createAxis(name,ar)

Create an Axis in a Dataset. **(This function is not yet implemented.)**

CdmsFile.createAxis(name,ar,unlimited=0)

Create an Axis in a CdmsFile.

name is the string name of the Axis.

ar is a 1-D data array which defines the Axis values. It may have the value None if an unlimited axis is being defined.

At most one Axis in a CdmsFile may be designated as being 'unlimited', meaning that it may be extended in length. To define an axis as unlimited, either:

- set ar to None, and leave *unlimited* undefined, or
- set ar to the initial 1-D array, and set *unlimited* to **cdms.Unlimited**

Table 2.8 Axis Methods

Type	Method Definition
Array	**array = axis[i:j]** Read a slice of data from the external dataset. Data is returned in the physical ordering defined in the dataset. See Table 2.9 on page 37 for a description of slice operators.
None	**axis[i:j] = array** Write a slice of data to the external dataset. **(axes in CdmsFiles only)**
None	**assignValue(array)** Set the entire value of the axis. *array* is a one-dimensional, Numeric array.
None	**designateCircular(modulo, persistent=0)** Designate the axis to be circular. *modulo* is the modulus value. Any given axis value x is treated as equivalent to x+modulus If *persistent* is true, the external file or dataset (if any) is modified. By default, the designation is temporary.
None	**designateLatitude(persistent=0):** Designate the axis to be a latitude axis. If *persistent* is true, the external file or dataset (if any) is modified. By default, the designation is temporary.

Table 2.8 Axis Methods

Type	Method Definition
None	**designateLevel(persistent=0)** Designate the axis to be a vertical level axis. If *persistent* is true, the external file or dataset (if any) is modified. By default, the designation is temporary.
None	**designateLongitude(persistent=0, modulo=360.0)** Designate the axis to be a longitude axis. *modulo* is the modulus value. Any given axis value x is treated as equivalent to x+modulus If *persistent* is true, the external file or dataset (if any) is modified. By default, the designation is temporary.
None	**designateTime(persistent=0, calendar = cdtime.GregorianCalendar)** Designate the axis to be a time axis. If *persistent* is true, the external file or dataset (if any) is modified. By default, the designation is temporary. *calendar* is defined as in **getCalendar**().
Array	**getBounds**() Get the boundary array, or None if no bounds are defined.

Table 2.8 **Axis Methods**

Type	Method Definition
Integer	**getCalendar()** Returns the calendar associated with the (time) axis. Possible return values, as defined in the cdtime module, are: • cdtime.GregorianCalendar: the standard Gregorian calendar • cdtime.JulianCalendar: years divisible by 4 are leap years • cdtime.NoLeapCalendar: a year is 365 days • cdtime.Calendar360: a year is 360 days • None: no calendar can be identified Note: If the axis is not a time axis, the global, file-related calendar is returned.
Array	**getValue()** Get the entire axis vector.
Integer	**isCircular()** Returns true if the axis has circular topology. An axis is defined as circular if: • axis.topology=='circular', or • axis.topology is undefined, and the axis is a longitude The default cycle for circular axes is 360.0
Integer	**isLatitude()** Returns true iff the axis is a latitude axis.
Integer	**isLevel()** Returns true iff the axis is a level axis.

Table 2.8 Axis Methods

Type	Method Definition
Integer	**isLinear()** Returns true iff the axis has a linear representation.
Integer	**isLongitude()** Returns true iff the axis is a longitude axis.
Integer	**isTime()** Returns true iff the axis is a time axis.
Integer	**len(axis)** The length of the axis.

Table 2.8 **Axis Methods**

Type	Method Definition
Tuple	**mapInterval(interval)** Map a coordinate interval to an index interval. *interval* is a tuple having one of the forms: (x,y) (x,y,indicator) (x,y,indicator,cycle) None where *x* and *y* are coordinates indicating the interval [x,y), and: *indicator* is a two-character string, where the first character is 'c' if the interval is closed on the left, 'o' if open, and the second character has the same meaning for the right-hand point. (Default: 'co') If *cycle* is specified, the axis is treated as circular with the given cycle value. By default, if axis.isCircular() is true, the axis is treated as circular with a default value of 360.0. An interval of None returns the full index interval of the axis. The method returns the corresponding index interval [i,j), where i<j, indicating the half-open index interval i<=k<j, or None if the intersection is empty. Note: if the interval is interior to the axis, but does not span any axis element, a singleton (i,i+1) indicating an adjacent index is returned. For an axis which is circular (axis.topology == 'circular'), [i,j) is interpreted as follows (where N = len(axis)): • if j <= N, the interval does not wrap around the axis endpoint

Table 2.8 Axis Methods

Type	Method Definition
None	**setBounds(bounds, persistent=0)** Set the boundary array. *bounds* is a NumPy array bnds of shape (len(axis),2), such that the boundaries of the nth axis value are [bnds[n,0],bnds[n,1]]. Note: By default, the boundaries are not written to the file or dataset containing the axis (if any). This allows bounds to be set on read-only files, for regridding. If the optional argument *persistent* is set to 1, the boundary array is written to the file.
None	**setCalendar(calendar, persistent=1)** Set the calendar for this (time) axis. *calendar* is defined as in getCalendar(). If *persistent* is true, the external file or dataset (if any) is modified. This is the default.
Axis	**subaxis(i,j)** Create an axis associated with the integer range [i:j]. The result axis is not associated with a file or dataset.
String	**typecode()** The Numeric datatype identifier.

Table 2.9 Axis Slice Operators

Slice	Definition
[i]	The ith element, starting with index 0
[i:j]	The ith element through, but not including, element j
[i:]	The ith element through and including the end
[:j]	The beginning element through, but not including, element j

Table 2.9 **Axis Slice Operators**

Slice	Definition
[:]	The entire array
[i:j:k]	Every kth element, starting at i, through but not including j
[-i]	The ith element from the end. -1 is the last element.

Example: A longitude axis has value [0.0, 2.0, ..., 358.0], of length 180. Map the coordinate interval -5.0 <= x < 5.0 to index interval(s), with wrap-around. The result index interval 178<=k<183 wraps around, since 180<183. This is equivalent to the two index intervals 178<=k<180 and 0<=k<3

```
> axis.isCircular()
1
> axis.mapInterval((-5.0,5.0))
(178,183)
>
```

2.6 *CdmsFile*

A CdmsFile is a physical file, accessible via the cdunif interface. netCDF files are accessible in read-write mode. All other formats (DRS, HDF, GrADS/GRIB, POP, QL) are accessible read-only.

Table 2.10 **CdmsFile Internal Attributes**

Type	Name	Definition
Dictionary	attributes	Global, external file attributes
Dictionary	axes	Axis objects contained in the file.
Dictionary	grids	Grids contained in the file.
String	id	File pathname.
Dictionary	variables	Variables contained in the file.

Table 2.11 CdmsFile Constructors

cdms.openDataset(path, mode)

Open the file specified by path.

path is the file pathname, a string.

mode is the open mode indicator, as listed in Table 2.16 on page 43.

cdms.createDataset(path)

Create the file specified by *path*, a string.

Table 2.12 CdmsFile Methods

Type	Definition
None	**close()** Close the file.
Axis	**copyAxis(axis, newname=None)** Copy axis values and attributes to a new axis in the file. The returned object is persistent: it can be used to write axis data to or read axis data from the file. If an axis already exists in the file, having the same name and coordinate values, it is returned. It is an error if an axis of the same name exists, but with different coordinate values. *axis* is the axis object to be copied. *newname*, if specified, is the string identifier of the new axis object. If not specified, the identifier of the input axis is used.

Table 2.12 CdmsFile Methods

Type	Definition
Grid	**copyGrid(grid, newname=None)** Copy grid values and attributes to a new grid in the file. The returned grid is persistent. If a grid already exists in the file, having the same name and axes, it is returned. An error is raised if a grid of the same name exists, having different axes. *grid* is the grid object to be copied. *newname*, if specified is the string identifier of the new grid object. If unspecified, the identifier of the input grid is used.
Axis	**createAxis(String id, Array ar, Integer unlimited=0)** Create a new Axis. This is a persistent object which can be used to read or write axis data to the file. *ar* is the one-dimensional axis array. Set *unlimited* to cdms.Unlimited to indicate that the axis is extensible.
RectGrid	**createRectGrid(id, lat, lon, order, type="generic", mask=None)** Create a RectGrid in the file. This is not a persistent object: the order, type, and mask are not written to the file. However, the grid may be used for regridding operations. *lat* is a latitude axis in the file. *lon* is a longitude axis in the file. *order* is a string with value "yx" (the first grid dimension is latitude) or "xy" (the first grid dimension is longitude). *type* is one of 'gaussian','uniform','equalarea',or 'generic' If specified, *mask* is a two-dimensional, logical Numeric array (all values are zero or one) with the same shape as the grid.

Table 2.12 CdmsFile Methods

Type	Definition
Variable	**createVariable(String id, String datatype,List axes)** Create a new Variable. This is a persistent object which can be used to read or write variable data to the file. *id* is a String name which is unique with respect to all other objects in the file. *datatype* is a CDMS datatype, as listed in Table 2.13 on page 41. *axes* is a list of Axis objects.
Variable	**createVariableCopy(var, newname=None)** Create a new Variable, with the same name, axes, and attributes as the input variable. An error is raised if a variable of the same name exists in the file. *var* is the Variable to be copied. *newname*, if specified is the name of the new variable. If unspecified, the returned variable has the same name as *var*. Note: Unlike copyAxis, the actual data is not copied to the new variable.
None	**sync()** Writes any pending changes to the file.

Table 2.13 CDMS Datatypes

CDMS Datatype	Definition
CdChar	character
CdDouble	double-precision floating-point
CdFloat	floating-point
CdInt	integer

Table 2.13 CDMS Datatypes

CDMS Datatype	Definition
CdLong	long integer
CdShort	short integer

2.7 Database

2.8 Dataset

A Dataset is a virtual file. It consists of a metafile, in CDML/XML representation, and one or more data files.

Table 2.14 Dataset Internal Attributes

Type	Name	Summary
Dictionary	attributes	Dataset external attributes.
Dictionary	axes	Axes contained in the dataset.
Dictionary	grids	Grids contained in the dataset.
String	id	Dataset identifier.
String	mode	Open mode.
String	uri	Uniform Resource Identifier of this dataset.
Dictionary	variables	Variables contained in the dataset.
Dictionary	xlinks	External links contained in the dataset.

Table 2.15 Dataset Constructors

cdms.openDataset(String uri, String mode='r')

Open the dataset specified by the Universal Resource Indicator, a CDML file. mode is one of the indicators listed in Table 2.16 on page 43.

cdms.createDataset(String path, String directory, String fileTemplate)

(Note: this function is not yet implemented)

Create a new dataset. *path* is the filepath of a CDML file. *fileTemplate* describes how the dataset is to be partitioned. It is a pathname, relative to the directory, which contains zero or more template specifiers (see Table 2.17 on page 44). A template may contain directory names as well as file names. A template specifier is a string of the form '%X' or '%eX', where X is one of the characters listed Table 2.17 on page 44. The form '%eX' may be used to specify the end time or level value. A specifier may appear more than once in a template.

Table 2.16 Open Modes

Mode	Definition
'r'	read-only
'r+'	read-write
'a'	read-write. Open the file if it exists, otherwise create a new file
'w'	Create a new file, read-write

Table 2.18 Dataset Methods

Type	Definition
None	**close()** Close the dataset.
	createRectGrid(id, lat, lon, order, type="generic", mask=None) Create a RectGrid in the dataset. This is not a persistent object: the order, type, and mask are not written to the dataset. However, the grid may be used for regridding operations. *lat* is a latitude axis in the dataset. *lon* is a longitude axis in the dataset. *order* is a string with value "yx" (the first grid dimension is latitude) or "xy" (the first grid dimension is longitude). *type* is one of 'gaussian','uniform','equalarea',or 'generic' If specified, *mask* is a two-dimensional, logical Numeric array (all values are zero or one) with the same shape as the grid.
List	**getPaths()** Get a sorted list of pathnames of datafiles which comprise the dataset. This does not include the XML metafile path, which is stored in the **.uri** attribute.
None	**sync()** Write any pending changes to the dataset.

2.9 *RectGrid*

A RectGrid is a two-dimensional, horizontal, rectilinear grid. A rectGrid can be defined in terms of a pair of axes, one longitude and one latitude. A two-dimensional, logical mask array may optionally be associated with a rectGrid.

Table 2.19 RectGrid Internal Attributes

Type	Name	Definition
Dictionary	attributes	External attribute dictionary.
String	id	The grid identifier.
Dataset or CdmsFile	parent	The dataset or file which contains the grid.
Tuple	shape	The shape of the grid, a 2-tuple.

Table 2.20 RectGrid Constructors

cdms.createRectGrid(lat, lon, order, type="generic", mask=None)

Create a grid not associated with a file or dataset.

See Table 2.2 on page 18.

CdmsFile.createRectGrid(id, lat, lon, order, type="generic", mask=None)

Create a grid associated with a file. See Table 2.12 on page 39.

Dataset.createRectGrid(id, lat, lon, order, type="generic", mask=None)

Create a grid associated with a dataset. See Table 2.18 on page 45.

Table 2.21 RectGrid Methods

Type	Definition
Axis	**getAxis(Integer n)** Get the n-th axis. n is either 0 or 1.
Tuple	**getBounds()** Get the grid boundary arrays. Returns a tuple (latitudeArray, longitudeArray), where latitudeArray is a Numeric array of latitude bounds, with shape (n,2), and longitudeArray is a similar array for longitude bounds. If no boundary arrays are explicitly defined (in the file or dataset), the result depends on the autoBounds mode (see **cdms.setAutoBounds**) and the grid classification mode (see **cdms.setClassifyGrids**). By default, autoBounds mode is enabled, in which case the boundary arrays are generated based on the type of grid. If disabled, the return value is (None,None). The grid classification mode specifies how the grid type is to be determined. By default, the grid type (Gaussian, uniform, etc.) is determined by calling **grid.classifyInFamily.** If the mode is 'off' **grid.getType** is used instead
Axis	**getLatitude()** Get the latitude axis of this grid.
Axis	**getLongitude()** Get the latitude axis of this grid.

Table 2.21 **RectGrid Methods**

Type	Definition
Array	**getMask()** Get the mask array of this grid, if any. Returns a 2-D Numeric array, having the same shape as the grid. If the mask is not explicitly defined, the return value is None.
String	**getOrder()** Get the grid ordering, either "yx" if latitude is the first axis, or "xy" if longitude is the first axis.
String	**getType()** Get the grid type, either "gaussian", "uniform", "equalarea", or "generic".

(Array,Array)	**getWeights()**
	Get the normalized area weight arrays, as a tuple (latWeights, lonWeights). It is assumed that the latitude and longitude axes are defined in degrees.
	The latitude weights are defined as:
	`latWeights[i] = 0.5 * abs(sin(latBounds[i+1]) - sin(latBounds[i]))`
	The longitude weights are defined as:
	`lonWeights[i] = abs(lonBounds[i+1] - lonBounds[i])/360.0`
	For a global grid, the weight arrays are normalized such that the sum of the weights is 1.0
	Example: Generate the 2-D weights array, such that weights[i.j] is the fractional area of grid zone [i,j].
	`import Numeric`
	`latwts, lonwts = grid.getWeights()`
	`weights = latwts[:,Numeric.NewAxis]*lonwts`

Table 2.21 RectGrid Methods

Type	Definition
None	**setBounds(latBounds, lonBounds, persistent=0)** Set the grid boundaries. *latBounds* is a NumPy array of shape (n,2), such that the boundaries of the kth axis value are [*latBounds*[k,0],*latBounds*[k,1]]. *lonBounds* is defined similarly for the longitude array. Note: By default, the boundaries are not written to the file or dataset containing the grid (if any). This allows bounds to be set on read-only files, for regridding. If the optional argument *persistent* is set to 1, the boundary array is written to the file.
None	**setMask(mask, persistent=0)** Set the grid mask. If persistent==1, the mask values are written to the associated file, if any. Otherwise, the mask is associated with the grid, but no I/O is generated. *mask* is a two-dimensional, Boolean-valued Numeric array, having the same shape as the grid.
None	**setType(gridtype)** Set the grid type. *gridtype* is one of "gaussian", "uniform", "equalarea", or "generic".

Table 2.21 RectGrid Methods

Type	Definition
RectGrid	**subGrid((latStart,latStop),(lonStart,lonStop))** Create a new grid, with latitude index range [latStart : latStop] and longitude index range [lonStart : lonStop]. Either index range can also be specified as None, indicating that the entire range of the latitude or longitude is used. For example, `newgrid = oldgrid.subGrid(None, (lonStart, lonStop))` creates `newgrid` corresponding to all latitudes, and index range [lonStart.lonStop] of `oldgrid`. If a mask is defined, the subgrid also has a mask corresponding to the index ranges. Note: The result grid is not associated with any file or dataset.

Table 2.21 RectGrid Methods

Type	Definition
RectGrid	**subGridRegion(latInterval, lonInterval)** Create a new grid corresponding to the coordinate region defined by latInterval, lonInterval. *latInterval* and *lonInterval* are the coordinate intervals for latitude and longitude, respectively. Each interval is a tuple having one of the forms: (x,y) (x,y,indicator) (x,y,indicator,cycle) None where *x* and *y* are coordinates indicating the interval [x,y), and: *indicator* is a two-character string, where the first character is 'c' if the interval is closed on the left, 'o' if open, and the second character has the same meaning for the right-hand point. (Default: 'co') If *cycle* is specified, the axis is treated as circular with the given cycle value. By default, if grid.isCircular() is true, the axis is treated as circular with a default value of 360.0. An interval of None returns the full index interval of the axis. If a mask is defined, the subgrid also has a mask corresponding to the index ranges. Note: The result grid is not associated with any file or dataset.

Table 2.21 RectGrid Methods

Type	Definition
RectGrid	**transpose()** Create a new grid, with axis order reversed. The grid mask is also transposed. Note: The result grid is not associated with any file or dataset.

2.10 Variable

A Variable is a multidimensional data object. The domain of a variable is defined in terms of Axis and Grid objects.

A Variable is contained in a Dataset. Setting a slice of a Variable writes data to the Dataset, and referencing a Variable slice reads data from the Dataset.

Table 2.22 Variable Internal Attributes

Type	Name	Definition
Dictionary	attributes	External attribute dictionary.
List	domain	Axes contained in the dataset. Each element of the list is itself a tuple of the form `(axis,start,length,true_length)` where *axis* is an axis object, *start* is the start index of the domain relative to the axis object, *length* is the length of the axis, and *true_length* is the actual number of (defined) points in the domain.
String	id	Variable identifier.
String	name_in_file	The name of the variable in the file or files which represent the dataset. If different from **id**, the variable is 'aliased'.

Table 2.22 Variable Internal Attributes

Type	Name	Definition
Dataset or CdmsFile	parent	The dataset or file which contains the variable.
Tuple	shape	The length of each axis of the variable.

Table 2.23 Variable Constructors

Dataset.createVariable(String id, String datatype, List axes)

Create a Variable in a Dataset. **This function is not yet implemented.**

CdmsFile.createVariable(String id, String datatype, List axesOrGrids)

Create a Variable in a CdmsFile.

id is the name of the variable.

datatype is a CDMS datatype, as defined in Table 2.13 on page 41..

axesOrGrids is a list of Axis and/or Grid objects, on which the variable is defined. Specifying a rectilinear grid is equivalent to listing the grid latitude and longitude axes, in the order defined for the grid.

Table 2.24 Variable Methods

Type	Definition
Array	**array = var[i:j, m:n]** Read a slice of data from the external dataset. Data is returned in the physical ordering defined in the dataset. The forms of the slice operator are listed in Table 2.25 on page 57. Note: enabling autoReshape mode causes singleton dimensions to be removed from the result array. (See **cdms.setAutoReshapeMode**.) By default, this mode is off.
	var[i:j, m:n] = array Write a slice of data to the external dataset. The forms of the slice operator are listed in Table 2.21 on page 32. (Variables in CdmsFiles only)
None	**assignValue(Array ar)** Write the entire data array. Equivalent to var[:] = ar. (Variables in CdmsFiles only).
Axis	**getAxis(n)** Get the n-th axis. *n* is an integer.
Grid	**getGrid()** Return the associated grid, or None if the variable is not gridded.
Axis	**getLatitude()** Get the latitude axis, or None if not found.
Axis	**getLevel()** Get the vertical level axis, or None if not found.

Table 2.24 Variable Methods

Type	Definition
Axis	**getLongitude()** Get the longitude axis, or None if not found.
Various	**getMissing()** Get the missing data value, or None if not found.
String	**getOrder()** Get the order string of a spatio-temporal variable. The order string specifies the physical ordering of the data. It is a string of characters with length equal to the rank of the variable, indicating the order of the variable's time, level, latitude, and/or longitude axes. Each character is one of: 't': time 'z': vertical level 'y': latitude 'x': longitude '-': the axis is not spatio-temporal. **Example**: A variable with ordering "tzyx" is 4-dimensional, where the ordering of axes is (time, level, latitude, longitude). Note: The order string is of the form required for the *order* argument of a regridder function.
List	**getPaths(*intervals)** Get the file paths associated with the index region specified by intervals. *intervals* is a list of scalars, 2-tuples representing [i,j), slices, and/or Ellipses. If no argument(s) are present, all file paths associated with the variable are returned. Returns a list of tuples of the form (path,slicetuple), where *path* is the path of a file, and *slicetuple* is a tuple of slices, of the same length as the rank of the variable, representing the region of the variable which is contained in the file.

Table 2.24 Variable Methods

Type	Definition
Array	**getRegion(*region)** Read a region of data. A region is a hyperrectangle in coordinate space. *region* is an argument list, each item of which specifies an interval of a coordinate axis. The intervals are listed in the order of the variable axes. If trailing dimensions are omitted, all values of those dimensions are retrieved. If an axis is circular (**axis.isCircular**() is true) or cycle is specified (see below), then data will be read with wraparound in that dimension. Only one axis may be read with wraparound. A coordinate interval has one of the forms listed in Table 2.26 on page 58. Also see **cdms.setAutoReshapeMode**. See examples below.
String	**getTemplate**() Get the file template associated with this variable. If no template is associated with the variable, the dataset template is returned.
Axis	**getTime**() Get the time axis, or None if not found.
Array	**getValue**() Read the entire array. Equivalent to a = var[:]
Integer	**len(var)** The length of the first dimension of the variable.
String	**typecode**() The Numeric datatype identifier.

Example: Get a region of data.

Variable ta is a function of (time, latitude, longitude). Read data corresponding to all times, latitudes -45.0 up to but not including +45.0, longitudes 0.0 through and including longitude 180.0:

```
data = ta.getRegion(':', (-45.0,45.0), (0.0, 180.0, 'cc'))
```

In the previous example, assume that times are represented as relative times with units "days since 1979-01-01". Read all data for 1980:

```
import cdtime

# Convert absolute times 1980-01-01, 1981-01-01 to
# relative times with the correct units.

t80 = cdtime.comptime(1980).torel("days since 1979")
t81 = cdtime.comptime(1981).torel("days since 1979")

# Read the data for 1980. The interval represents all
# times t such that t80 <= t < t81. Also note that
# intervals for the trailing dimensions latitude
# and time can be omitted.

data = ta.getRegion((t80,t81))
```

Table 2.25 Variable Slice Operators

[i]	The ith element, zero-origin indexing.
[i:j]	The ith element through, but not including, element j
[i:]	The ith element through the end
[:j]	The beginning element through, but not including, element j
[:]	The entire array
[i:j:k]	Every kth element
[i:j, m:n]	Multidimensional slice
[i, ..., m]	(Ellipsis) All values of all dimensions between the first and last
[-1]	Negative indices 'wrap around'. -1 is the last element.

Table 2.26 **Coordinate Intervals used in getRegion()**

Interval	Definition	Example
x	single point, such that axis[i]==x	180.0
(x,y)	indices i such that x <= axis[i] < y	(-180,180)
(x,y,'cc')	x <= axis[i] <= y The third item is defined as in mapInterval.	(-90,90,'cc')
(x,y,'co',cycle)	x<= axis[i] < y, with wraparound Note: It is not necesary to specify the cycle of a circular longitude axis, that is, for which axis.isCircular() is true.	(-180,180,'co',360.0)
':' or None	all axis values of one dimension	
Ellipsis	all values of all intermediate axes	

2.11 Examples

In this example, two datasets are opened, containing surface air temperature ('tas') and upper-air temperature ('ta') respectively. Surface air temperature is a function of (time, latitude, longitude). Upper-air temperature is a function of (time, level, latitude, longitude). Time is assumed to have a relative representation in the datasets (e.g., with units "months since basetime").

Data is extracted from both datasets for January of the first input year through December of the second input year. For each time and level, three quantities are calculated: slope, variance, and correlation. The results are written to a netCDF file. For brevity, the functions corrCoefSlope and removeSeasonalCycle are omitted.

(1)

```
import cdms, Numeric
```

```
from cdtime import *

# Write slope, correlation, and variance variables
def writeNetCDF(lons,lats,levs,file_name,title,b,c,v):
    file = cdms.createDataset(file_name + '.nc')
    file.title = title
    lon_var = file.createAxis('longitude', lons)
    lon_var.units = "degrees_east"
    lat_var = file.createAxis('latitude', lats)
    lat_var.units = "degrees_north"
    lev_var = file.createAxis('level',levs)
    lev_var.units = 'mb'

    foo = file.createVariable('slope', cdms.CdDouble, (lev_var, lat_var,
        lon_var))
    foo[:] = b
    foo = file.createVariable('correlation', cdms.CdDouble, (lev_var, lat_var,
        lon_var))
    foo[:] = c
    foo = file.createVariable('variance', cdms.CdDouble, (lev_var, lat_var,
        lon_var))
    foo[:] = v
    file.close()

def mapTimes(year1, year2, units, calendar):
    time1 = comptime(year1,1).torel(units,calendar).value
    time2 = comptime(year2,12).torel(units,calendar).value
    return time1,time2

# Calculate variance, slope, and correlation of surface air tempature
# with upper air temperature
# by level, and save to a netCDF file. 'pathTa' is the location of
# the CDMS dataset containing ta, 'pathTas' is the file with contains tas.
# Data is extracted from January of year1 through December of year2.
def ccSlopeVarianceBySeasonFiltNet(pathTa,pathTas,year1,year2):

    # Open the files for ta and tas

    fta  = cdms.openDataset(pathTa)
    ftas = cdms.openDataset(pathTas)

    # Get upper air temperature and axes

    taObj = fta.variables['ta']
    levs  = taObj.getLevel()[:]
    lats  = taObj.getLatitude()[:]
    lons  = taObj.getLongitude()[:]

    # Surface temperature times
    timeObj = ftas.axes['time']
    calendar = timeObj.getCalendar()
    if calendar==None: calendar=NoLeapCalendar

    # Get the timepoints corresponding to January of year1,
    # and December of year2.
    time1, time2 = mapTimes(year1,year2,timeObj.units,calendar)

    # Get the surface temperature for the closed interval [time1,time2]
    i1,i2 = timeObj.mapInterval((time1,time2),'cc')
    tas = ftas.variables['tas'][i1:i2]

    # assert time_bounds[0] == 1 and time_bounds[1] == 12
```

```
    cc = Numeric.zeros( (len(levs),tas.shape[1],tas.shape[2]), Numeric.Float)
    b = Numeric.zeros( (len(levs),tas.shape[1],tas.shape[2]), Numeric.Float)
    v = Numeric.zeros( (len(levs),tas.shape[1],tas.shape[2]), Numeric.Float)

    # Remove seasonal cycle from surface air temperature
    tas = removeSeasonalCycle(tas)

    # Get correct indices for ta
    timeObj = fta.axes['time']
    calendar = timeObj.getCalendar()
    if calendar==None: calendar=cdtime.NoLeapCalendar
    time1, time2 = mapTimes(year1,year2,timeObj.units,calendar)
    i1,i2 = timeObj.mapInterval((time1,time2),'cc')

    # For each level of air temperature, remove seasonal cycle
    # from upper air temperature, and calculate statistics
    for ilev in range(len(levs)):
        print 'level = ',ilev, levs[ilev]
        ta = taObj[i1:i2,ilev]
        ta.shape = tas.shape # Ensure that the arrays conform
        ta = removeSeasonalCycle(ta)
        cc[ilev], b[ilev]  = corrCoefSlope(tas ,ta)
        v[ilev] = Numeric.add.reduce( ta**2 )/(1.0*ta.shape[0])
    file_name = 'CC_B_V_ALL'
    title = 'filtered'
    writeNetCDF(lons,lats,levs,file_name,title,b,cc,v)

if __name__=='__main__':
    pathTa = '/pcmdi/cdms/sample/ccmSample_ta.xml'
    pathTas = '/pcmdi/cdms/sample/ccmSample_tas.xml'
    # Process Jan80 through Dec81
    ccSlopeVarianceBySeasonFiltNet(pathTa,pathTas,1980,1981)
```

(8)

(9)

Notes:

1. Three modules are imported, **cdms**, **Numeric**, and **cdtime**. **Numeric** implements array functions. **cdtime** supports time arithmetic.
2. The writeNetCDF function creates a new netCDF file, and writes three variables to the file: `b` (slope), `c` (correlation), and `v` (variance). `lons`, `lats`, and `levs` are 1-D arrays for longitude, latitude, and level axes, respectively.

   The file is created with the **createDataset** function. Since the file extension is not `.xml` or `.cdml`, a CdmsFile is created.

   Setting `file.title` creates and sets a global attribute in the file.

   Three axes are created via **createAxis**, and the units attributes are set.

   The three variables are created via **createVariable**. The domain is specified as a list of axis objects created previously.

The line `foo[:] = b` writes the array b to variable foo in the file.

It is important to close the file, to ensure that all data is written.

3. `mapTimes` returns a tuple of relative time values (`time1`, `time2`), where:
   - `time1` is January of `year1`, and
   - `time2` is December of `year2`.
   comptime is a **cdtime** function which creates a component time. **torel**() translates to the appropriate relative units.
4. The two datasets are opened via **openDataset**(). `fta` is the dataset containing upper-air temperature, and `ftas` is the dataset containing surface air temperature.
5. The variable `taObj` is retrieved using the predefined dataset attribute `.variables`. This is a dictionary with the variable ids as keys.

   getLevel() returns the level (vertical dimension) axis for `ta`. The slice operator [:] reads the entire array, so that `levs` is a Numeric array containing the levels. The same is true of latitude and longitude.
6. Datasets have a `.axes` attribute, which is a dictionary of all Axes in the file. It is assumed that the time axis has id `'time'`, so `timeObj` is the time axis. A better approach is to use the **getTime**() function to retrieve the time axis.

   `calendar` is the **cdtime** calendar associated with the time axis. If no calendar is specified in the dataset, it is assumed to be the Gregorian calendar.
7. The **mapInterval** function maps the coordinate interval (`time1, time2`) to an index interval. The optional `'cc'` indicator specifies that the interval is closed on both ends, that is , `time1` and `time2` are both contained in the interval. If the indicator were omitted, it would default to `'co'`, meaning closed on the left, open on the right.

   mapInterval returns indices `(i1,i2)`, which represents all integers k such that i1<=k<i2. In other words, the closed coordinate interval `(time1,time2,'cc')` maps to the half-open index interval `(i1,i2)`.

   This could also have been accomplished more directly using the **getRegion**() function, which takes an argument list of coordinate intervals. The following obtains the same result:

```
tasObj = ftas.variables['tas']
tas = tasObj.getRegion((time1,time2,'cc'))
```

8. `ta` is read using a multidimensional slice operator. Since `ta` is assumed to be a function of (time, level, latitude, longitude), the operation
   `ta = taObj[i1:i2,ilev]`
   reads times with indices `i1` through `i2-1`, level `ilev`, all latitudes, all longitudes.
9. This is the main routine of the script. `pathTa` and `pathTas` are dataset paths which reference the XML metafiles. Data is processed from January 1980 through December 1981.

In the next example, the pointwise variance of a variable over time is calculated, for all times in a dataset. The name of the dataset is input, all variables in the dataset are printed, then the name of the variables is selected. The variance is then calculated and plotted via the vcs module.

```
#!/pcmdi/drach/cdat/python15/python
#
# Calculates gridpoint total variance
# from an array of interest
#

from Numeric import *
import cdms

AxisNotTime = 'First axis is not time, variable:'

# Create a netCDF file, write v(lon,lat)
def writenc(filename,lons,lats,v):
    f = cdms.createDataset('calcVar.nc')
    lon = f.createAxis('longitude',lons)
    lat = f.createAxis('latitude',lats)
    varvar = f.createVariable('variance', cdms.CdDouble, (lat,lon))
    varvar[:] = v
    f.close()

# Generate a plot of a 2-D array 'ar'.
# 'w' is the VCS window object returned from vcs.init()
# 'xar' and 'yar' are the x-axis and y-axis coordinates.
# 'aname', 'xname', and 'yname' are the names of the array, x-axis, and y-axis.
# 'xbounds' and 'ybounds' are boundary arrays.
(10) def plot2d (w,ar,xar,yar,aname,xname,yname,units=None,
            xbounds=None,ybounds=None):
    if xbounds is None:
       xbounds = [1.5*xar[0]-0.5*xar[1]] + ((xar[0:-1]+xar[1:])/2.0).tolist() +
         [1.5*xar[-1]-0.5*xar[-2]]
    if ybounds is None:
       ybounds = [1.5*yar[0]-0.5*yar[1]] + ((yar[0:-1]+yar[1:])/2.0).tolist() +
         [1.5*yar[-1]-0.5*yar[-2]]
    ar.setdimattribute(0,'values',yar.tolist())
    ar.setdimattribute(1,'values',xar.tolist())
    ar.setdimattribute(0,'bounds',ybounds)
    ar.setdimattribute(1,'bounds',xbounds)
    ar.setdimattribute(0,'name',yname)
    ar.setdimattribute(1,'name',xname)
```

```
    if units is not None:
        ar.createattribute('units')
        ar.setattribute('units',units)
    ar.setattribute('name',aname)
    w.plot(ar,'AMIP')

# Wait for return in an interactive window
def pause():
    print 'Hit return to continue: ',
    line = sys.stdin.readline()

# Calculate pointwise variance of variable over time
# Returns the variance and the number of points
# for which the data is defined, for each grid point

def calcVar(var):
    # Check that the first axis is a time axis
(11) firstaxis = var.domain[0][0]
    if not firstaxis.isTime():
        raise AxisNotTime, var.id

    # Read the entire variable
    x = var[:]

    n = 1.*avg.count(x)
    sumxx = addmissing.reduce(x*x)
    sumx = addmissing.reduce(x)
    variance = (n*sumxx - (sumx * sumx))/(n * (n-1.))

    return variance,n

if __name__=='__main__':
    import vcs, sys

    print 'Enter dataset path [/pcmdi/cdms/sample/obs/erbs_mo.xml]: ',
    path = string.strip(sys.stdin.readline())
    if path=='': path='/pcmdi/cdms/sample/obs/erbs_mo.xml'

    # Open the dataset
(12) dataset = cdms.openDataset(path)

    # Select a variable from the dataset
    print 'Variables in file:',path
    varnames = dataset.variables.keys()
    varnames.sort()
    for varname in varnames:
        var = dataset.variables[varname]
        if hasattr(var,'long_name'):
            long_name = var.long_name
        elif hasattr(var,'title'):
            long_name = var.title
        else:
            long_name = '?'
        print '%-10s: %s'%(varname,long_name)
    print 'Select a variable: ',
    varname = string.strip(sys.stdin.readline())
    var = dataset.variables[varname]

    # Calculate variance
    variance,n = calcVar(var)
    dataset.close()
```

```
# Get longitude and latitude arrays
x = var.getLongitude()[:]
y = var.getLatitude()[:]

# Save the data
writenc('calcVar.nc',x,y,variance)

# Plot variance
w=vcs.init()
w.setcolormap('default')
if hasattr(var,'units'):
    units = var.units
else:
    units = None
plot2d(w,variance,x,y,varname+'
     variance','longitude','latitude','(%s)^2'%units)
pause()
w.clear()
plot2d(w,n,x,y,varname+' npts defined','longitude','latitude')
pause()
w.clear()
```

The result of running this script is as follows:

```
% calcVar.py
Enter dataset path [/pcmdi/cdms/sample/obs/erbs_mo.xml]:
 Variables in file: /pcmdi/cdms/sample/obs/erbs_mo.xml
albt       : Albedo TOA [%]
albtcs     : Albedo TOA clear sky [%]
rlcrft     : LW Cloud Radiation Forcing TOA [W/m^2]
rlut       : LW radiation TOA (OLR) [W/m^2]
rlutcs     : LW radiation upward TOA clear sky [W/m^2]
rscrft     : SW Cloud Radiation Forcing TOA [W/m^2]
rsdt       : SW radiation downward TOA [W/m^2]
rsut       : SW radiation upward TOA [W/m^2]
rsutcs     : SW radiation upward TOA clear sky [W/m^2]
Select a variable: albt

<The variance is plotted>

Hit return to continue:

<The number of points is plotted>
```

Notes:

10. The plot2d function creates a boxfill plot of the 2-D array **ar** in window **w**. The setdimattribute and setattribute functions are PCMDI Numeric extensions.
11. The domain of a variable is a list [elem1, elem2, ..., elemn] where each element is a tuple of the form (axis,start,length,true_length). In this example, **var.domain[0]** is the domain element for the first axis, and **var.domain[0][0]** is the first axis object.
12. The dataset is opened via **openDataset**().

13. `var.getLongitude()` gets the longitude axis. The slice operator [:] reads the associated array.
14. The variance is plotted first, then the number of defined points is plotted.

CHAPTER 3

Regridding data

3.1 Overview

This chapter describes how to interpolate gridded CDMS data to another horizontal grid, within Python.

Regridding data is a two-step process:

- Given an input grid and output grid, generate a regridder function.
- Call the regridder function on a Numeric array, resulting in an array defined on the output grid.

The following example illustrates this process. The regridder function is generated at line 9, and the regridding is performed at line 10:

```
 1  #!/usr/local/bin/python
 2  import cdms
 3  from regrid import Regridder
 4  f = cdms.openDataset('/pcmdi/cdms/exp/cmip2/ccc/perturb.xml')
 5  rlsf = f.variables['rls']
 6  ingrid = rlsf.getGrid()
 7  g = cdms.openDataset('/pcmdi/cdms/exp/cmip2/mri/perturb.xml')
 8  outgrid = g.variables['rls'].getGrid()
 9  regridfunc = Regridder(ingrid, outgrid)
10  rlsnew = regridfunc(rlsf[:])
```

```
11  f.close()
12  g.close()
```

Line	Notes
2	Makes the CDMS module available.
3	Makes the `Regridder` class available from the `regrid` module.
4	Opens the input dataset.
5	Gets the variable object named `'rls'`. No data is read.
6	Gets the input grid.
7	Opens a dataset to retrieve the output grid.
8	The output grid is the grid associated with the variable named `'rls'` in dataset `g`. Just the grid is retrieved, not the data.
9	Generates a regridder function `regridfunc`.
10	Reads all data for variable `rlsf`, and calls the regridder function on that data, resulting in a Numeric array `rlsnew`.

3.2 *regrid module*

The regrid module implements the regridding functionality. Although this module is not strictly a part of CDMS, it is designed to work with CDMS objects. The Python command

```
from regrid import Regridder
```

makes the Regridder class available within a Python program. An instance of Regridder is a function which regrids data from input to output grid.

Table 3.1 **Regridder Constructor**

regridFunction = Regridder(inputGrid, outputGrid)

Create a regridder function which interpolates a data array from input to output grid. Table 3.2 on page 71 describes the calling sequence of this function.

inputGrid and *outputGrid* are CDMS grid objects.

Note: To set the mask associated with *inputGrid* or *outputGrid*, use the grid **setMask** function.

3.3 *regridder functions*

A regridder function is an instance of the Regridder class. The function is associated with an input and output grid. Typically its use is straightforward: the function is passed an input array and returns the regridded array. However, when the array has missing data, or the input and/or output grids are masked, the logic becomes more complicated.

Step 1: The regridder function first forms an *input mask*. This mask is either two-dimensional or 'n-dimensional', depending on the rank of the user-supplied mask.

Two-dimensional case:

- Let *mask_1* be the two-dimensional user mask supplied via the `mask` argument, or the mask of the input grid if no user mask is specified.
- If a missing-data value is specified via the `missing` argument, let the *implicit_mask* be the two-dimensional mask defined as 0 where the first horizontal slice of the input array is missing, 1 elsewhere.
- The input mask is the logical AND(*mask_1*, *implicit_mask*)

N-dimensional case: If the user mask is 3 or 4-dimensional with the same shape as the input array, it is used as the input mask.

Step 2: The data is then regridded. In the two-dimensional case, the input mask is 'broadcast' across the other dimensions of the array. In other words, it assumes that all horizontal slices of the array have the same mask. The result is a new array, defined on the output grid. Optionally, the regridder function can also return an array having the same shape as the output array, defining the fractional area of the output array which overlaps a non-missing input grid cell. This is useful for calculating area-weighted means of masked data.

Step 3: Finally, if the output grid has a mask, it is applied to the result array. Where the output mask is 0, data values are set to the missing data value, or 1.0e20 if undefined.

Table 3.2 Regridder function

Type	
Array	***regridFunction*(array, missing=None, order=None, mask=None)** Interpolate a gridded data array to a new grid. The interpolation preserves the area-weighted mean on each horizontal slice. An array of the same rank as the input array is returned. *array* is a Numeric array of rank 2, 3, or 4. *missing* is a Float specifying the missing data value. The default is 1.0e20. *order* is a string indicating the order of dimensions of the array. It has the form returned from **variable.getOrder()**. For example, the string "tzyx" indicates that the dimension order of *array* is (time, level, latitude, longitude). If unspecified, the function assumes that the last two dimensions of *array* match the input grid. *mask* is a Numeric array, of datatype Integer or Float, consisting of ones and zeros. A value of 0 or 0.0 indicates that the corresponding data value is to be ignored for purposes of regridding. If *mask* is two-dimensional of the same shape as the input grid, it overrides the mask of the input grid. If the mask has more than two dimensions, it must have the same shape as *array*. In this case, the *missing* data value is also ignored. Such an n-dimensional mask is useful if the pattern of missing data varies with level (e.g., ocean data) or time.
Array, Array	***regridFunction*(ar, missing=None, order=None, mask=None, returnTuple=1)** If called with the optional returnTuple argument equal to 1, the function returns a tuple (*dataArray*, *maskArray*). *dataArray* is the result data array. *maskArray* is a Float32 array of the same shape as *dataArray*, such that *maskArray*[i,j] is fraction of the output grid cell [i,j] overlapping a non-missing cell of the input grid.

3.4 Examples

Example: Create a uniform output grid.

```
1  #!/usr/local/bin/python
2  import cdms
3  from regrid import Regridder
4  f = cdms.openDataset('rls_ccc_per.nc')
5  rlsf = f.variables['rls']
6  ingrid = rlsf.getGrid()
7  outgrid = cdms.createUniformGrid(90.0, 46, -4.0, 0.0, 72, 5.0)
8  regridFunc = Regridder(ingrid, outgrid)
9  newrls = regridFunc(rlsf[:], missing=rlsf.getMissing())
10 f.close()
```

Line	Notes
4	Open a netCDF file for input.
7	Create a 4 x 5 degree output grid. Note that this grid is not associated with a file or dataset
8	Create the regridder function
9	Read all data and regrid. The missing data flag is set explicitly.

Example: Get a mask from a separate file, and set as the input grid mask.

```
1  import cdms
2  from regrid import Regridder
3  cdms.setAutoReshapeMode('on')
4  f = cdms.openDataset('so_ccc_per.nc')
5  sof = f.variables['so']
6  ingrid = sof.getGrid()
```

```
 7  g = cdms.openDataset('rls_mri_per.nc')
 8  rlsg = g.variables['rls']
 9  outgrid = rlsg.getGrid()
10  regridFunc = Regridder(ingrid,outgrid)
11  h = cdms.openDataset('sft_ccc.nc')
12  sfth = h.variables['sft']
13  sftArray = sfth[:]
14  outArray =
    regridFunc(sof[:],missing=sof.getMissing(),mask=sftMask)
15  f.close()
16  g.close()
17  h.close()
```

Line	Notes
3	Enable autoreshape mode. This removes singleton dimensions when data is read from a file.
6	Get the input grid.
9	Get the output grid
10	Create the regridder function.
13	Get the mask.
14	Regrid with a user mask. The same thing could be accomplished by setting the mask of ingrid via the **setMask** method. Note: Although it cannot be determined from the code, both `mask` and the input array `sof` are four-dimensional. This is the 'n-dimensional' case.

Example: Generate an array of zonal mean values.

```
1  f = cdms.openDataset('rls_ccc_per.nc')
2  rlsf = f.variables['rls']
3  ingrid = rlsf.getGrid()
4  outgrid = cdms.createZonalGrid(ingrid)
5  regridFunc = Regridder(ingrid,outgrid)
```

```
6  mean = regridFunc(rlsf[:])
7  f.close()
```

Line	Notes
3	Get the input grid.
4	Create a zonal grid. `outgrid` has the same latitudes as `ingrid`, and a singleton longitude dimension. **createGlobalMeanGrid** could be used here to generate a global mean array.
5	Generate the regridder function.
6	Generate the zonal mean array.

Example: Regrid an array with missing data, and calculate the area-weighted mean of the result.

```
1  from Numeric import *
   ...
2  outgrid = cdms.createUniformGrid(90.0, 46, -4.0, 0.0, 72, 5.0)
3  outlatw, outlonw = outgrid.getWeights()
4  outweights = outlatw[:,NewAxis]*outlonw
5  grid = var.getGrid()
6  sample = var[0,0]
7  latw, lonw = grid.getWeights()
8  weights = latw[:,NewAxis]*lonw
9  inmask = where(greater(absolute(sample),1.e15),0,1)
10 mean = add.reduce(ravel(inmask*weights*sample))/
   add.reduce(ravel(inmask*weights))
11 regridFunc = Regridder(grid, outgrid)
12 outsample, outmask = regridFunc(sample, mask=inmask,
   returnTuple=1)
13 outmean = add.reduce(ravel(outmask*outweights*outsample))/
   add.reduce(ravel(outmask*outweights))
```

Line	Notes
2	Create a uniform target grid.
3	Get the latitude and longitude weights.
4	Generate a 2-D weights array.
5	Get the input grid. **`var`** is a 4-D variable.
6	Get the first horizontal slice from **`var`**.
7-8	Get the input weights, and generate a 2-D weights array.
9	Set the 2-D input mask.
10	Calculate the input array area-weighted mean.
11	Create the regridder function.
12	Regrid. Because **`returnTuple`** is set to 1, the result is a tuple (dataArray, maskArray).
13	Calculate the area-weighted mean of the regridded data. **`mean`** and **`outmean`** should be approximately equal.

CHAPTER 4

Plotting CDMS data in Python

4.1 Overview

Data read via the CDMS Python interface can be plotted using the **vcs** module. This module, part of the Climate Data Analysis Tool (CDAT) is documented in the CDAT reference manual. The **vcs** module provides access to the functionality of the VCS visualization program.

Examples of plotting data accessed from CDMS are given below, as well as documentation for the **plot** routine keywords.

4.2 Examples

In the following examples, it is assumed that variable `psl` is dimensioned (time, latitude, longitude). `psl` is contained in the dataset named `'sample.xml'`.

4.2.1 Example: plotting a horizontal grid

```
1  import cdms, vcs
2  cdms.setAutoReshapeMode('on')
```

```
3  f = cdms.openDataset('sample.xml')
4  psl = f.variables['psl']
5  sample = pr[0]
6  w=vcs.init()
7  w.setcolormap('default')
8  w.plot(sample, variable=psl)
9  f.close()
```

Notes:

Line	Notes
2	Remove singleton dimensions when data is read.
5	Get a horizontal slice, for the first timepoint.
6	Create a VCS Canvas `w`.
7	Set the default colormap.
8	Plot the data. By default, a boxfill plot of a horizontal lat-lon array is generated. The variable `psl` encapsulates information on the grid coordinates, variable name, units, etc.
9	Close the file. This must be done after the reference to the persistent variable `psl`.

That's it! The axis coordinates, variable name, description, units, etc. are obtained from variable `psl`.

What if the units are not explicitly defined for `psl`, or a different description is desired? **plot** has a number of other keywords.

4.2.2 Example: using plot keywords.

```
w.plot(sample, variable=psl, units='mm/day', file_comment='High-
    frequency reanalysis', long_name="Sea level pressure",
    comment1="Sample plot", hms="18:00:00", ymd="1978/01/01")
```

Notes:

- Keyword arguments can be listed in any order.
- Specific keywords take precedence over general keywords. In this example, the units `'mm/day'` takes precedence over `psl.units`.

4.2.3 Example: plotting a time-latitude slice

If the data to be plotted is not a lat-lon slice, the xaxis and yaxis keywords are used to specify the axes:

```
    ...
1   samp = psl[:,:,0]
2   lat = psl.getLatitude()
3   time = psl.getTime()
4   w = vcs.init()
5   w.plot(samp, name='psl', xaxis=lat, yaxis=time)
```

Notes:

Line	Notes
1	`samp` consists of all times, latitudes for longitude index 0
2	`lat` is the CDMS latitude axis object, not just the array. The **xarray/ yarray** keywords can be used to specify a 1-D Numeric vector of values, as an alternative. The advantage of using **xaxis** and **yaxis** is that the **plot** routine can recognize the spatial orientation of the axes.
5	The **variable** keyword was not used here, so the **name** keyword defines the identifier.

4.2.4 Example: plotting subsetted data

It is important to note that a data array read from CDMS does not carry spatial coordinate information or other metadata with it, with the exception of a missing data value. The array argument of **plot** is just Numeric array, which can be read from a file or generated by a Numeric operation. There may not be a persistent variable or axis associated with the data a priori.

In the following example, the data corresponds to a proper subset of the time axis. A new CDMS axis object is created, corresponding to the subset retrieved.

```
   ...
1  samp = psl[0:100,:,0]
2  lat = psl.getLatitude()
3  time = psl.getTime()
4  w = vcs.init()
5  w.plot(samp, name='psl', xaxis=lat, yaxis=time.subaxis(0,100))
```

Because the first 100 times are retrieved, `samp` does not correspond to the dataset time axis, which contains all the time values. The **subaxis** method creates a new axis object corresponding to the first 100 timepoints.

4.3 plot method

The **plot** method is documented in the CDAT Reference Manual. This section augments the documentation with a description of the optional keyword arguments.

The general form of the plot command is:

canvas.**plot**(*array* [, *args*] [,*key*=*value* [, *key*=*value* [, ...]]])

where:

- *canvas* is a VCS Canvas object, created with the **vcs.init** method.
- *array* is a Numeric array, having between two and five dimensions. The last dimensions of the array is termed the 'x' dimension, the next-to-last the 'y' dimension, then 'z', 't', and 'w'. For example, if the array is three-dimensional,

the axes are (z,y,x). If array is four-dimensional, the axes are (t,z,y,x), and so on. (Note that the 't' dimension need have no connection with time; any spatial axis can be mapped to any plot dimension. For a graphics method which is two-dimensional, such as boxfill, the y-axis is plotted on the horizontal, and the x-axis on the vertical.

- *args* are optional positional arguments:

  args := *template_name, graphics_method, graphics_name*
  template_name: the name of the VCS template (e.g., 'AMIP')
  graphics_method : the VCS graphics method ('boxfill')
  graphics_name: the name of the specific graphics method ('default')

  See the CDAT Reference Manual and VCS Reference Manual for a detailed description of these arguments.

- *key=value*, ... are optional keyword/value pairs, listed in any order. These are defined in Table 4.1 on page 81.

Table 4.1 plot keywords

key	type	value
comment1	string	Comment plotted above file_comment
comment2	string	Comment plotted above comment1
comment3	string	Comment plotted above comment2
continents	0 or 1	if ==1, plot continental outlines (default: plot if xaxis is longitude, yaxis is latitude -or- xname is 'longitude' and yname is 'latitude'
file_comment	string	Comment, defaults to variable.parent.comment)

Table 4.1 **plot keywords**

key	type	value
grid	CDMS grid object	Grid associated with the data. Defaults to variable.getGrid()
hms	string	Hour, minute, second
long_name	string	Descriptive variable name, defaults to variable.long_name.
missing_value	same type as array	Missing data value, defaults to variable.getMissing()
name	string	Variable name, defaults to variable.id
time	cdtime relative or absolute time	time associated with the data. Example: cdtime.reltime(30.0, "days since 1978-1-1")
units	string	Data units. Defaults to variable.units
variable	CDMS variable object	Variable associated with the data. The variable grid must have the same shape as the data array.
xarray ([y|z|t|w]array)	1-D Numeric array	Array of coordinate values, having the same length as the corresponding dimension. Defaults to xaxis[:] (y|z|t|waxis[:])
xaxis ([y|z|t|w]axis)	CDMS axis object	Axis object. **xaxis** defaults to grid.getAxis(0), **yaxis** defaults to grid.getAxis(1)

Table 4.1 **plot keywords**

key	type	value
xbounds (ybounds)	2-D Numeric array	Boundary array of shape (n,2) where n is the axis length. Defaults to xaxis.getBounds(), or xaxis.genGenericBounds() if None, similarly for **ybounds**.
xname ([y|z|t|w]name)	string	Axis name. Defaults to xaxis.id ([y|z|t|w]axis.id)
xrev (yrev)	0 or 1	If **xrev (yrev)** is 1, reverse the direction of the x-axis (y-axis). Defaults to 0, with the following exceptions: • If the y-axis is latitude, and has decreasing values, yrev defaults to 1 • If the y-axis is a vertical level, and has increasing pressure levels, yrev defaults to 1.
xunits ([y|z|t|w]units)	string	Axis units. Defaults to xaxis.units ([y|z|t|w]axis.units).
xweights (yweights)	1-D Numeric array	Axis weights, a NumPy array of the same length as the dimension, used to calculate the area-weighted mean. This keyword is meaningful only if the data is a horizontal, lat-lon slice. Defaults to grid.getWeights().

CHAPTER 5

Climate Data Markup Language (CDML)

5.1 Introduction

The Climate Data Markup Language (CDML) is the language used to represent metadata in CDMS. CDML is based on the W3C XML standard.

CHAPTER 6

CDMS Utilities

6.1 cdimport: Importing data into CDMS

T

V